(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
25 January 2001 (25.01.2001)

PCT

(10) International Publication Number
WO 01/06357 A1

(51) International Patent Classification[7]: G06F 9/44

(21) International Application Number: PCT/US99/16770

(22) International Filing Date: 26 July 1999 (26.07.1999)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
09/353,394 15 July 1999 (15.07.1999) US

(71) Applicant *(for all designated States except US)*: AMERICAN MANAGEMENT SYSTEMS, INCORPORATED [US/US]; 4050 Legato Road, Fairfax, VA 22033 (US).

(72) Inventors; and
(75) Inventors/Applicants *(for US only)*: HOHMANN, Andreas [DE/DE]; Elfgenweg 14, D-40547 Düsseldorf (DE). DUYMELINCK, Erik [FR/DE]; Kaiserswertherstrasse 70, D-40477 Düsseldorf (DE).

(74) Agent: BECKERS, J., Randall; Staas & Halsey LLP, Suite 500, 700 11th Street, Washington, DC 20001 (US).

(81) Designated States *(national)*: AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, UA, UG, US, UZ, VN, YU, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, SD, SL, SZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published:
— *With international search report.*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

WO 01/06357 A1

(54) Title: META-LANGUAGE FOR C++ BUSINESS APPLICATIONS

(57) Abstract: A C++ meta-language ("CML") scripting language implemented on top of C++ providing direct coupling to C++ using a reflection layer. This CML scripting language provides for a high-level object-based scripting language with simple syntax. It also provides for a strongly typed with type inference capability. Using this CML scripting language compilation into internal representation for a stack machine is first accomplished. A Reflection framework (RTTI) allows for direct access to C++ class attributes and methods by name and makes possible for the script to interact with business objects programmed in C++.

META-LANGUAGE FOR C++ BUSINESS APPLICATIONS

Reference To Microfiche Appendix

A microfiche appendix having 3 microfiche and 262 frames is included herewith containing the detailed design specification for the present invention.

BACKGROUND OF THE INVENTION

Field of the Invention

The present invention is directed to a meta-language for C++ business software applications. More particularly, to a C++ meta-language that can access C++ objects and interface to a run-time type identification mechanism that enables the changing of a C++ meta-language scripts without the need to recompile C++ source code or for the user to be familiar with a programming language.

Description of the Related Art

In the past in order to improve the coding, debug and maintenance of business logic high level languages such as COBOL were developed specifically for business applications. However, a

skilled programmer in consultation with a person knowledgeable with the application was needed. Development and debug times were long and modification of programs was difficult and expensive.

Meta-languages were developed to eliminate the programmer from the loop, but did not offer access to C++ objects.

Therefore, what is needed is a meta-language that someone without computer programming skills can use and run without recompilation of core C++ application software. Further, what is needed is a C++ meta-language that can be used to configure business logic at run time and which can utilize logic implemented in C++ business objects.

Existing scripting languages include JavaScript, Tcl, Python, ML (Meta Language), VisualBasic and Perl. However, no existing scripting language provides the features of being compiled, strongly typed, type inference, object oriented with direct access to C++ objects along with a reflection layer bound to the C++ language which allows for modification of business logic without recompilation being necessary.

SUMMARY OF THE INVENTION

It is an object of the present invention to provide a C++ Meta-Language ("CML") capable of directly interfacing to C++ objects and thereby use the full power and flexibility of an object oriented language.

It is also an object of the present invention for a person with no programming skills or limited programming skills to be able to generate a CML script that can perform complex business applications.

It is still a further object of the present invention to allow for

scripts to be changed and executed without the need for recompilation of the C++ source code.

It is also an object of the present invention to allow for fast execution of generated scripts.

The above objects can be attained by a system and method that provides a C++ Meta-Language ("CML") for C++ business applications. The CML provides a scripting layer on top of and interfacing to a C++ layer. The use of a meta-language allows to control the functionality at run-time, using conditions and processing rules, stored in reference data or plain text files. Business functionality, implemented using the CML, can be changed without recompilation of C++ code and due to the higher level of abstraction of the CML, non-technical staff can maintain meta-language scripts.

These together with other objects and advantages which will be subsequently apparent, reside in the details of construction and operation as more fully hereinafter described and claimed, reference being had to the accompanying drawings forming a part hereof, wherein like numerals refer to like parts throughout.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 is an overall software configuration of the present invention.

Figure 2 is a diagram showing the interaction between the CML source script, the CML Framework and the business application in the present invention.

Figure 3 is a flowchart showing the method of creating and executing a CML script as shown in figure 2 as practiced by an embodiment of the present invention.

Figure 4 is a diagram showing the architecture of the stack-

machine in the present invention.

DESCRIPTION OF THE PREFERRED EMBODIMENTS

Before discussing the features of the present invention a summary of the terms used in the discussion herein will be provided.

A scripting language is a high-level language providing basic programming constructs.

A meta-language is a high-level language that is directly tied to a specific application.

Reference data is usually stored on a permanent storage such as a database and is used by processing the data as reference values for validation or conditional logic.

Business logic is the implementation of a business process using rules and processing logic. If a business process or associated rules need to be changed, the business logic must also be modified.

A class is a program module that combines logic and state. An example of ~~an~~ class is the Event class that is used to represent telephone calls.

An object is a specific instance of a class. An example of an object is the instance of a Event object that represents a specific telephone call. The Event object is an instance of the Event class, its state represents the data of the specific telephone call.

An attribute is part of an object's state and defines some property of the class. An example of an attribute in an Event object is the originating phone number of the telephone call that is represented by the object.

A method is part of a class or object and performs specific operations. Methods can be called by other objects or by the object itself. An example of a method in an Event object is a method

providing the call duration. The duration can either be calculated (from start and end time) or simply provide access to a duration attribute in the Event object.

A reflection layer provides a mechanism to browse objects at runtime for their type, attributes and methods. This facilitates late binding where it is determined at runtime what operations are executed.

A strongly typed system allows the use of typed data in programs to be checked when the program is compiled. For example it can be verified whether a return value of a certain type can be assigned to a variable of a certain type.

A type inference, in a language providing type inference, the types of literals, values, and expressions in a program do not have to be declared by the programmer but are calculated once the program is compiled.

In a compiled language, the program text has to be compiled before execution. This results in a performance improvement and the execution is less prone to contain errors. All syntactical and type related information can be verified by the compiler to prevent errors at execution time.

The C++ Meta-Language (CML) framework 34 shown in Figure 2 comprises compiler 120 and stack machine 100 and provides applications with tools necessary to compile and execute meta-language scripts. The CML framework is coupled with a C++ reflection layer, also referred to as run-time type identification ("RTTI") 32 mechanism shown in figure 1. All components of the CML framework are located on an application server and are implemented in portable C++. The operating systems used are HP-UX, Linux, and Windows NT. However, as would be appreciate by someone of

ordinary skill in the art, any general purpose computer operating system may be used. Using another high level program language other than C++, would also require the implementation of the RTTI system and the CML compiler/stack machine libraries, since they are tightly coupled to C++.

Software layers

Referring to Figure 1, without the existence of the meta-language layer 10, all business application logic would be traditionally implemented in the application core layer 20 in form of C++ business objects. In order to change to the business logic, C++ code would have to be modified, the application core layer would have to be recompiled and re-tested. Non-technical staff, familiar with the business process, could not perform these modifications. However, with meta-language layer 10, business logic can be implemented in the meta-language layer 10 by the non-technical staff. The meta-language 10 provides a higher level of abstraction than C++ and can more easily be maintained by non-technical staff. Business logic contained in meta-language 10 can be modified without the extensive effort of rebuilding the application. A drawback of the meta-language layer 10 is performance. Processing implemented in meta-language 10 is executed more slowly than processing implemented in application core 20 C++ objects. Therefore, a balance of flexibility and performance needs to be considered when planning where to place business logic.

Meta-Language Layer 10

Still referring to Figure 1, meta-language layer 10 contains the specification of the business logic. This meta-language layer 10

covers the functionality that is subject to frequent change such as the output formats or processing rules. The meta-language layer 10 is based on building blocks provided by the application core 20. Most of the complex logic of the business functions is contained in the meta-language layer 10.

Application Core 20

The application core layer 20 includes application-specific C++ classes. This application core layer 20 also contains the hard-coded business functionality that is not embedded in the meta-language layer 10. The application core layer 20 is built on top of the infrastructure layer 30 for the particular kind of application (batch, online, server) intended. Performance critical processing has to be carried out by low level C++ objects and is implemented in the application core layer 20.

Infrastructure Layer 30

The infrastructure layer 30 provides services that can be used by all applications. The infrastructure offers a standard interface to the operating system 40 including additional services such database access, logging, messaging, and profiling. The infrastructure layer 30 provides common services that can be used by all applications built on top of the infrastructure. The infrastructure layer 30 also provides a reflection layer (RTTI) 32 that allows for access to attributes and methods of C++ objects by name as required by the scripting layer. Any C++ class can export its attributes and methods to be visible for CML, under certain names by which the attributes and methods can be accessed from CML. A reflection layer is a conventional mechanism for changing structure and behavior of software systems

dynamically. The reflection layer provides information and access to system properties and makes the software self-aware. Programming languages, such as JAVA, contain reflection layers as a feature of the language itself. Implementation of reflection layers is described in A SYSTEM OF PATTERNS, by Frank Buschman et. al., Wiley and Sons, ISBN 0 471 95869 7, incorporated by reference herein. Further, the C++ meta-language (CML) framework 34 is part of the infrastructure layer with its compiler 120, shown in figure 2, and stack machine 100, shown in figure 2.

The CML framework 34 shown in Figure 1 uses a combination of techniques to provide the meta-language layer 10 on top of application core layer 20. The main features of the CML framework 34 comprises:

1. High-level object-based scripting language with simple syntax;
2. Strongly typed with type inference (less type declarations);
3. Compilation into internal representation for a stack machine ("virtual machine"); and
4. Integrated with a Reflection Layer (RTTI) 32 allowing access to C++ class attributes and methods by name.

CML is a strongly-typed compiled language, as illustrated in the discussion of table 1 below. The target of the compilation is not machine code, but a sequence of instructions for a small virtual machine (stack machine 100 shown in Figure 2). Referring to Figure 2, this stack machine program 80 (SMP 80) is contained in a C++ structure and executed by a stack machine 100 realized in C++ when executing the program. This approach ensures that most of the burden of translating the CML source script 60, shown in Figure 2,

can be carried out before actually executing the program. Since CML is strongly typed, the compiler 120, shown in Figure 2, can check the types of all operations avoiding type errors during the execution of the program. Moreover, the compiler can generate typed operations for the stack machine 100 that can be executed much faster than dynamically typed operations.

Sample Program

Table 1 below depicts a simple CML script 60, shown in figure 2, provided for illustrative purposes only dealing with a very simple example of bill formatting. The purpose of this script is to print all events contained in the given document sorted by start time. It also has to compute the total duration as well as the total charge grouped by "event group". As a final operation the sample program prints the results.

In the sample program shown in table 1, it is assumed that three C++ classes have been defined with run-time type information using the RTTI mechanism. These three C++ classes are the Document class, Invoice class, and Event class. The Document class contains a list of events which is simply called events. The Invoice class extends the Document class by an integer attribute number. An Event class has four attributes, the start time, the duration, the charge, and the group. Moreover, it provides a method getDestination which returns the destination (called number) as a string.

TABLE 1 - SAMPLE CML PROGRAM

```
0  import usage;
1
2  program printEvent(event: Event)
3    println "call at ", event.start, " to ", event.getDestination();
4    usage.process(event);
5
6  program processDocument(document: Document)
7    if document is Invoice then
8      println "This is invoice number ", document.number;
9    end;
10
11    totalDuration = 0;
12    longCalls = list of Event;
13    chargeGroups = map of (string, decimal);
14
15    sort document.events by start;
16
17    for event in document.events do
18      printEvent(event);
19      with event do
20        if duration > 100 then
21          longCalls.append(event);
22        end;
23        totalDuration += duration;
24        chargeGroups[group] += charge;
25      end;
21    end;
22
23    for key in chargeGroups.keys() do
24      println "total for ", key, " is ", chargeGroup[key];
25    end;
25    println "total duration is ", totalDuration;
```

Table 1 - Sample CML Program.

The following is a brief description of the functions performed by the CML script shown in table 1.

Line 0 of table 1 imports another CML package called "usage". This package contains a program called "process" which is called in line 4.

Lines 2-4 of table 1 define the first program printEvent which contains the print commands for a single event. The argument type declaration uses a Pascal-like style. Event is the C++ class supplied with run-time type information so that its attributes and methods can be used from CML as shown in Line 3.

Lines 7-8 of table 1 show how the Invoice number attribute can be used. A type-check condition such as "is Invoice" automatically changes the type of the variable within the associated block (here, the "then" block of the if-statement). In this way attributes of derived classes can be accessed without any additional type casting.

Line 11 of table 1 introduces a new variable. Variables do not have to be declared. They are created with the first assignment, and their type is the type of the value which is assigned to it (here an integer).

Line 12 of table 1 defines another new variable which is a list of events. The right hand side of the assignment creates this list. For container types such as lists, vectors, tuples and maps, the contained types have to be specified to allow for compile-time type checking.

Line 13 of table 1 defines a map (also called associative array) of decimals indexed by strings. For maps, both, the type of the keys and the type of the values, have to be provided.

Line 15 of table 1 shows the sort statement for lists. It is one of the CML commands which have been built in for convenience.

Line 17 of table 1 starts a loop through the list of event contained in the document. The syntax for the control statements

(loops, conditional statements, etc.) has been chosen to be short but readable (no begin-end, no curly brackets as in some programming languages).

Line 18 of table 1 calls the first program printEvent with the current event.

Line 19 of table 1 shows the with statement which (as in most scripting languages) allows for using attributes names directly ("duration" instead of "event.duration").

Line 24 of table 1 demonstrates another feature which has been introduced to reduce the CML code size: the += operator for maps. If the key (here "group") does not exist in the map, the entry is automatically created and initialized with zero before adding the right hand side of the += statement.

As demonstrated in this small example shown in table 1, the CML syntax has been designed to be easy to learn with the idea of "executable pseudo code". Therefore, a conventional syntax (in contrast to Lisp, Smalltalk) has been chosen. Since CML sits on top of the object-oriented language C++, attribute and method access are supported using a syntax similar to C++.

Referring to Figures 2 and 3, the execution of the CML script 60 in figure 2 and operation 200 in figure 3 relies on the use of a stack machine 100 that introduces a significant overhead when compared to direct machine instructions. However, the advantage provided is run-time configurability which makes possible the ability of changing business functionality without recompiling C++ code. This overhead is unavoidable unless techniques like just-in-time compilation are employed. Just-in-time compilation refers to a technique where interpreted code is translated into machine code as it

part of the translation process is the handling of types. The CML compiler 120 uses bindings and environments. A binding gives a path (such as "event.duration") a specific meaning (such as "field of type int"). More precisely, a binding maps a path to a variable, field (attribute of an object variable), method, or a CML program that is part of another CML module. An environment manages the bindings during the compilation process. Besides keeping the bindings, the environment also handles the scope of variables and the logic for "with" statements. The environment changes during the compilation as new variables are created or existing ones go out of scope. Those changes are kept in a history that allows for recovering the state of the environment. The history is implemented as stacks (detailed further below) for each bound variable and another stack containing the paths that have been changed. Once the Compiler 120 has finished compiling the CML Source Script 60, it passes the SMP 80 back to the Application 65. The Application 65 keeps this SMP 80 obtained from the compiler 120 in memory. As provided in operation 230 shown n figure 3. Whenever data needs to be processed, the application checks 240 whether the CML source script 60 was modified since it was compiled last. If it was not modified, the application calls the stack machine 100 with the SMP 80 and the input data 90. The stack machine 100 executes the SMP 80 and returns the output data 110 to the application. If the CML source script 60 was modified, the application loads the modified version of the script and recompiles it before executing it.

Virtual Stack Machine 100

Referring to Figure 4, like most virtual machines, the CML framework uses a stack 160 to hold temporary values and a frame

150 to hold variables accessible by an index. The virtual stack machine 100 operates by copying data between stack 160 and frame 150 or act on the stack 160 itself as depicted in Figure 4. For ease of implementation, frame 150 and stack 160 use vectors of generic values.

A stack machine program (SMP) 80 is a sequence of operations 140 which determine the actions carried out with stack 160, frame 150, and output stream 110.

The stack 160 is a LIFO system (Last-In, First-Out). Values are retrieved (or popped) from the stack in the reverse order they were added (or pushed) onto it.

The frame 150 is an indexed storage system, i.e., it can be viewed as a set of numbered cells. Moving a value from or to the frame requires to know the index number of the cell from which a value is read or to which it is written. From the CML point of view, the frame manages the variables used in a program.

In the context of a CML program calling other CML programs, a called program will always have its own local Frame 150. In fact, a Frame 150 is associated with one instance of a SMP 80 only. At startup, it only contains the objects that are the parameters of that program (Input Data 90 shown in figure 2).

The output stream 110 is the place where text is sent to for output. From a CML program, the output stream 110 is the target of print statements.

Alternate Embodiments

The procedures presented herein are not inherently related to any particular computer. In particular, various general-purpose machines may be used with programs described herein. The recommended configuration is a general purpose computer running a

UNIX operating system.

Further, any number of computer languages may be used. For example, Java may be used instead of C++. Different version of UNIX may also be used as well as any comparable operating system. Almost any processor or computer may be used such as a Sun computer.

The many features and advantages of the invention are apparent from the detailed specification and, thus, it is intended by the appended claims to cover all such features and advantages of the invention which fall within the true spirit and scope of the invention. Further, since numerous modifications and changes will readily occur to those skilled in the art, it is not desired to limit the invention to the exact construction and operation illustrated and described, and accordingly all suitable modifications and equivalents may be resorted to, falling within the scope of the invention.

What is claimed is:

1. A meta-language for generating scripts used to execute business logic, comprising:

a script to execute business logic implemented in an object oriented programming language having a plurality of class attributes, methods, type inferences, and strongly typed data structures; and

a reflection layer interfacing to the script to access the plurality of class attributes and methods by name.

2. A meta-language as recited in claim 1, wherein the object oriented programming language is C++.

3. A meta-language as recited in claim 1, wherein the script is a high level language providing basic programming constructs.

4. A meta-language as recited in claim 1, further comprising:

a parser to scan and parse the script for grammatical and data typing errors and report any errors to a user.

5. A meta-language as recited in claim 4, further comprising:

a compiler to compile the script after it has been scanned and parsed by the parser for grammatical and data typing errors and has been found to be error free and generate a stack machine program.

6. A meta-language as recited in claim 5, further comprising:

a stack machine to execute the stack machine program using data input by the user and generating output data for the user.

7. A meta-language as recited in claim 6, wherein the script may be modified by the user and executed by the stack machine without the need of recompilation by the on any application code due to the reflection layer accessing the plurality of class attributes and methods by name.

8. A meta-language as recited in claim 7, wherein the stack machine retrieves operations from the stack machine program and translates the operations for execution.

9. A meta-language as recited in claim 5, wherein the compiler generates a plurality of bindings that maps a path to a variables, fields, or methods in an operation of the stack machine program.

10. A meta-language for generating scripts used to execute business logic, comprising:

a script to execute business logic implemented in C++ object oriented programming language having a plurality of class attributes, methods, type inferences, and strongly typed data structures;

a reflection layer interfacing to the script to access the plurality

of objects, class attributes and methods by name;

a parser to scan and parse the script for grammatical and data typing errors and report any errors to a user;

a compiler to compile the script after it has been scanned and parsed by the parser for grammatical and data typing errors and has been found to be error free and generate a stack machine program; and

a stack machine to execute the stack machine program using data input by the user and generating output data for the user, wherein the script accesses objects, class attributes and method in the C++ object oriented programming language.

11. A method of generating, modifying and executing a meta-language program that performs business logic, comprising:

creating a script to execute business logic implemented in an object oriented programming language having a plurality of class attributes, methods, type inferences, and strongly typed data structures; and

generating a reflection layer interfacing to the script to access the plurality of class attributes and methods by name.

12. The method as recited in claim 11, wherein the object oriented programming language is C++.

13. The method as recited in claim 11, wherein the script is a

high level language providing basic programming constructs.

14. The method as recited in claim 11, further comprising:

scanning and parsing the script for grammatical and data typing errors and reporting any errors to a user.

15. The method as recited in claim 14, further comprising:

compiling the script after it has been scanned and parsed by the parser for grammatical and data typing errors and has been found to be error free and generating a stack machine program.

16. The method as recited in claim 15, further comprising:

executing the stack machine program using data input by the user and generating output data for the user.

17. The method as recited in claim 16, wherein the script may be modified by the user and executed without the need of compilation due to accessing the plurality of class attributes and methods by name.

18. The method as recited in claim 17, further comprising:

retrieving operations from the stack machine program and translating the operations for execution.

19. The method as recited in claim 15, further comprising:

generating a plurality of bindings that maps a path to a plurality of variables, fields, or methods in an operation of the stack machine program.

20. A method of generating, modifying and executing a meta-language program that performs business logic, comprising:

creating a script to execute business logic implemented in a C++ object oriented programming language having a plurality of class attributes, methods, type inferences, and strongly typed data structures;

generating a reflection layer interfacing to the script to access the plurality of objects, class attributes and methods by name.;

scanning and parsing the script for grammatical and data typing errors and reporting any errors to a user;

compiling the script after it has been scanned and parsed by the parser for grammatical and data typing errors and has been found to be error free and generating a stack machine program;

executing the stack machine program using data input by the user and generating output data for the user;

retrieving operations from the stack machine program and translating the operations for execution; and

generating a plurality of bindings that maps a path to a plurality of variables, fields, or methods in an operation of the stack machine program.

21. A computer program stored on a computer readable medium for generating scripts used to execute business logic, comprising:

a script to execute business logic implemented in an object oriented programming language having a plurality of class attributes, methods, type inference, and strongly typed data structures; and

a reflection layer module interfacing to the script to access the plurality of class attributes and methods by name.

22. The computer program as recited in claim 21, wherein the object oriented programming language is C++.

23. The computer program as recited in claim 21, wherein the script is a high level language providing basic programming constructs.

24. The computer program as recited in claim 21, further comprising:

a parser module to scan and parse the script for grammatical and data typing errors and report any errors to a user.

25. The computer program as recited in claim 24, further comprising:

a compiler module to compile the script after it has been scanned and parsed by the parser module for and grammatical and

data typing errors and has been found to be error free and to generate a stack machine program.

26. The computer program as recited in claim 25, further comprising:

a stack machine to execute the stack machine program using data input by the user and generating output data for the user.

27. The computer program as recited in claim 26, wherein the script may be modified by the user and executed by the stack machine without the need of compilation by the compiler module due to the reflection layer accessing the plurality of objects, class attributes and methods by name.

28. The computer program as recited in claim 27, wherein the stack machine retrieves operations from the stack machine program and translates the operations for execution.

29. The computer program as recited in claim 25, wherein the compiler module generates a plurality of bindings that maps a path to a variables, fields, or methods in an operation of the stack machine program.

30. A computer program stored on a computer readable medium for generating scripts used to execute business logic,

comprising:

a script to execute business logic implemented in a C++ object oriented programming language having a plurality of class attributes, methods, type inference, and strongly typed data structures;

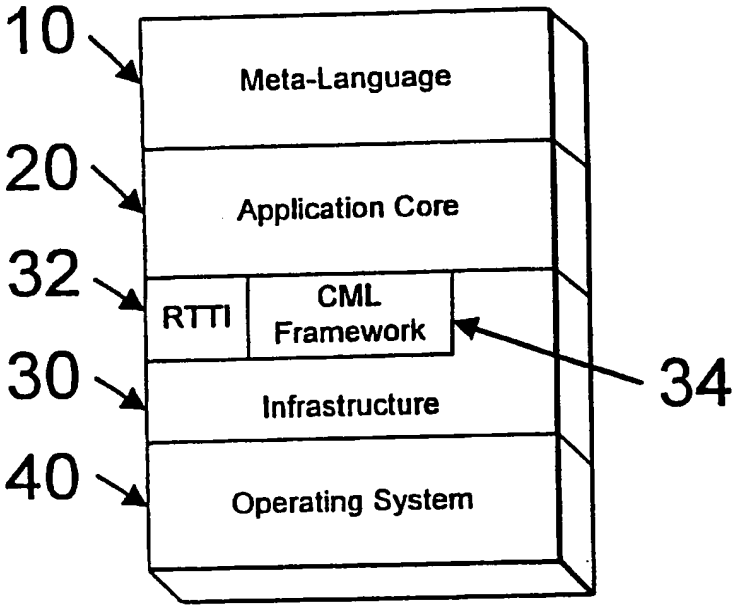
a reflection layer module interfacing to the script to access the plurality of objects, class attributes and methods by name.;

a parser module to scan and parse the script for grammatical and data typing errors and report any errors to a user;

a compiler module to compile the script after it has been scanned and parsed by the parser module for grammatical and data typing errors and has been found to be error free and to generate a stack machine program;

a stack machine to execute the stack machine program using data input by the user and generating output data for the user;

FIG. 1



Software Layers

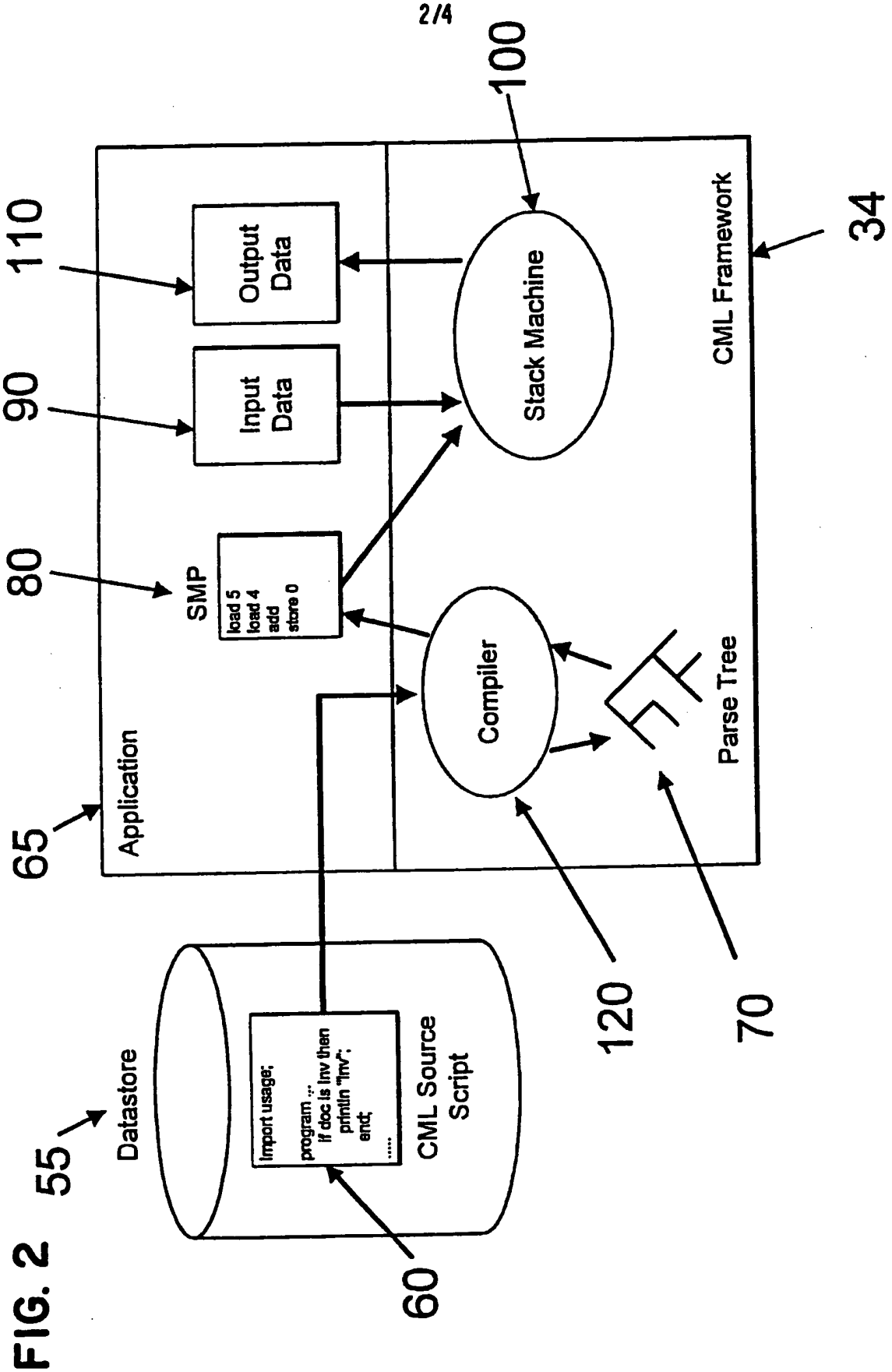
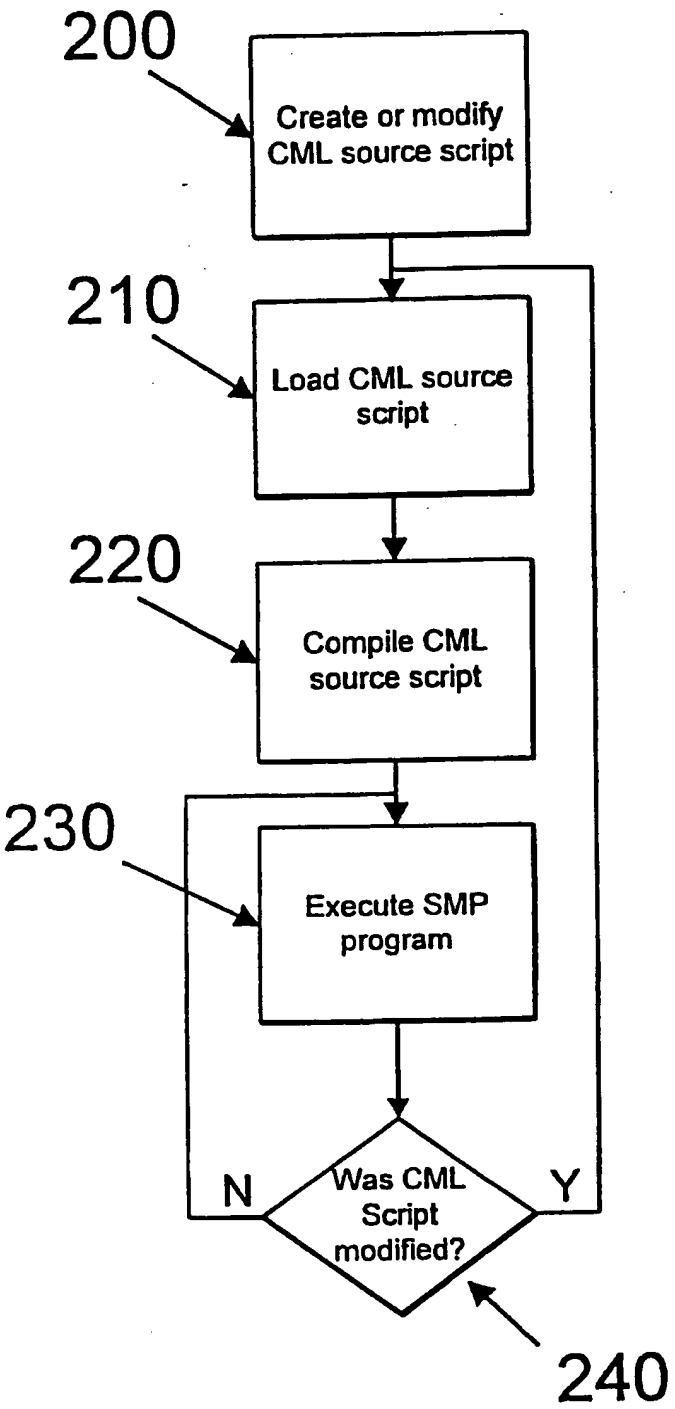

100
110
Output Data
90
Input Data
Stack Machine
CML Framework
34
80
SMP
load 5
load 4
add
store 0
Compiler
Parse Tree
65
Application
55
Datastore
import usage;
program ...
if doc is inv then println "inv";
end;
.....
CML Source Script
120
70
60

FIG. 2

FIG. 3


200
Create or modify CML source script
210
Load CML source script
220
Compile CML source script
230
Execute SMP program
N
Was CML Script modified?
Y
240

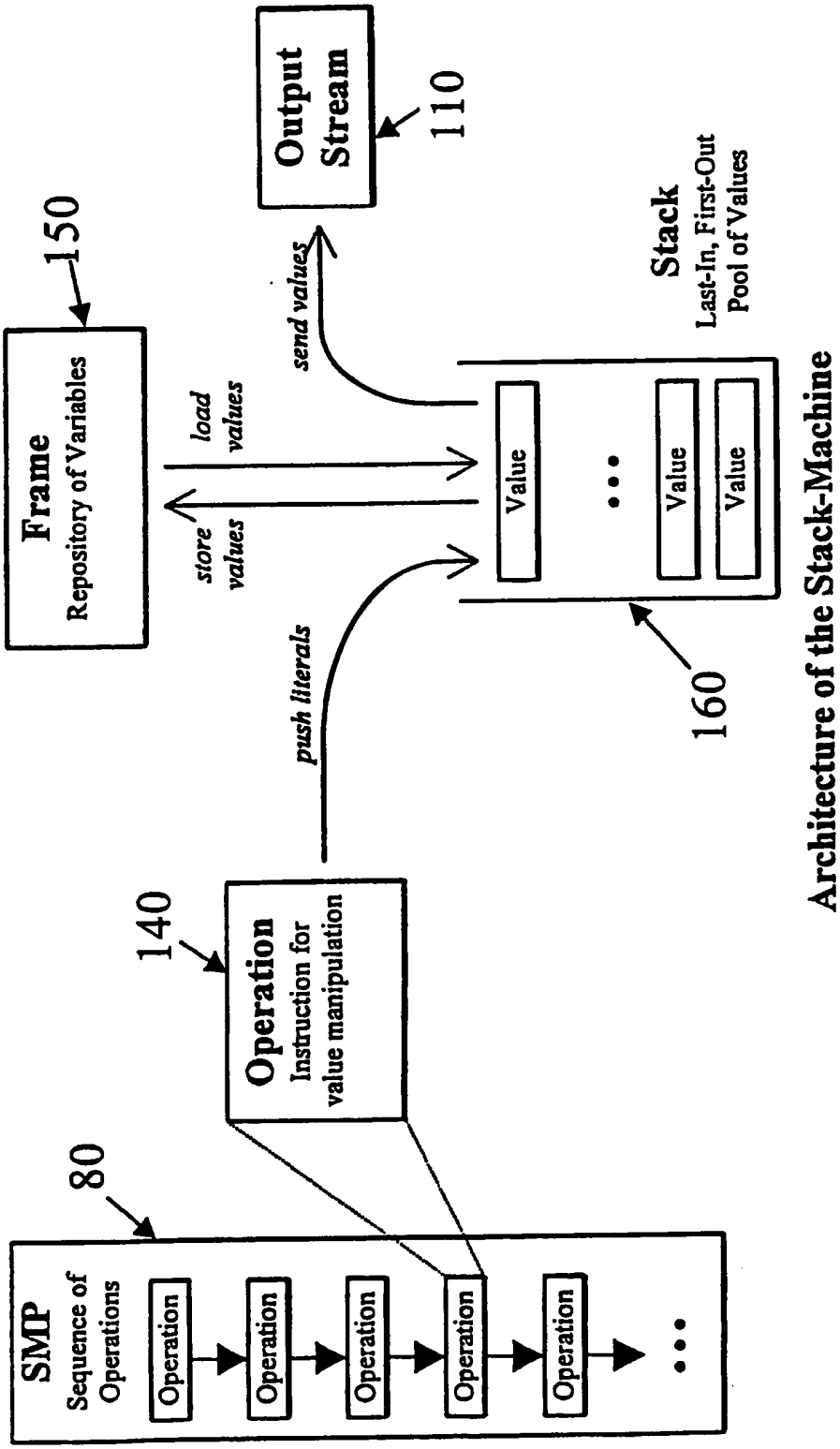


FIG. 4

Architecture of the Stack-Machine

INTERNATIONAL SEARCH REPORT

Int ational Application No
PCT/US 99/16770

A. CLASSIFICATION OF SUBJECT MATTER
IPC 7 G06F9/44

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 7 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	HOEVE VAN F ET AL: "AN OBJECT-ORIENTED APPROACH TO APPLICATION GENERATION" SOFTWARE PRACTICE & EXPERIENCE,GB,JOHN WILEY & SONS LTD. CHICHESTER, vol. 17, no. 9, 1 September 1987 (1987-09-01), pages 623-645, XP000001704 ISSN: 0038-0644 the whole document	1-30
A	GB 2 320 111 A (JBA HOLDINGS PLC) 10 June 1998 (1998-06-10) page 5, line 1 - line 15	1-30
A	EP 0 722 140 A (INST OF SOFTWARE SCIENT CONST ;JR EAST JAPAN INFORMATION SYST (JP)) 17 July 1996 (1996-07-17) page 2, line 34 -page 3, line 23	1-30

-/--

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

° Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
14 June 2000	21/06/2000

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Brandt, J

Form PCT/ISA/210 (second sheet) (July 1992)

INTERNATIONAL SEARCH REPORT

Int. .tional Application No
PCT/US 99/16770

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	BREUER P T ET AL: "A PRETTIER COMPILER-COMPILER: GENERATING HIGHER-ORDER PARSERS IN C" SOFTWARE PRACTICE & EXPERIENCE,GB,JOHN WILEY & SONS LTD. CHICHESTER, vol. 25, no. 11, 1 November 1995 (1995-11-01), pages 1263-1297, XP000655547 ISSN: 0038-0644 page 1270, line 29 - line 43 page 1289, line 6 -page 1293, line 1 page 1295, line 3 - line 20 -----	1-30

1

INTERNATIONAL SEARCH REPORT

Information on patent family members

Inte ional Application No
PCT/US 99/16770

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
GB 2320111 A	10-06-1998	AU 5401498 A	29-06-1998
		CN 1245571 A	23-02-2000
		EP 0943126 A	22-09-1999
		WO 9825203 A	11-06-1998
EP 0722140 A	17-07-1996	US 5809304 A	15-09-1998
		WO 9602033 A	25-01-1996